Harvey Mudd College
Computer Science 60
Fall 2007
Assignment 1
Functional Programming using Scheme
Due 11:59 p.m., Monday, 10 September 2007

All problems in this assignment are to be done in a purely functional style using the Scheme language. Submit a single file named hw01.scm containing all solutions using the web page http://www.cs.hmc.edu/~cs60grad/submissions/. **Be sure to spell each function name exactly as given**, otherwise the automatic grading script might not give you credit for the function. For this assignment, each problem 1-10 has the same point value.

Some test cases are indicated for each of the problems. These can be downloaded from the web copy of this handout. However, if you use auxiliary functions, it would be good practice to include test for them as well, and you might even want to devise additional tests for the main functions, so that your solution is more likely to pass any test. Unless mentioned, you may assume that the arguments have the form indicated in the problem statement. That is, your function does not have to validate its arguments.

0. Obtain access to Dr. Scheme, either by downloading it to your personal computer or by other means. Use the "Pretty Big" subset of Scheme as the language for this assignment. Type in a header for your file and the sample problem solution at the end of this document. Document each of your files, solutions, and functions in this way, throughout the course. We take off for lack of documentation.

1. Construct a function `eval-quadratic` that evaluates a quadratic expression $(ax + b)x + c$, given arguments a, b, c, and x in that order. Examples:

```
(eval-quadratic 1 2 3 4)
→ 27
(eval-quadratic 1 2 2 -1+1i)
→ 0
```

   (Note that Scheme supports complex numbers, such as –1+1i above. Also, scheme differentiates between exact numbers, such as integers or rationals, and inexact numbers, such as floating point. Complex numbers can be created from either exact or inexact. Do not use inexact constants when exact ones will do, otherwise your results will always be inexact.)

2. Construct a function `solve-quadratic1` that solves a quadratic given the coefficients a, b, c (i.e. finds one of the two solutions to $(ax + b)x + c = 0$). Assume that a is non-zero. Use this form of the quadratic formula: $(\text{sqrt}(b_2–4ac) –b)/2a$ (translated into Scheme of course). Use the Scheme built-in function sqrt. Examples:

```
(solve-quadratic1 1 -4 4)
→ 2
(solve-quadratic1 1 2 2)
→ -1+1i
(solve-quadratic1 1 1e10 1)
→ 0.0
```

3. Construct a function `test-quadratic-solver` that tests a function of the form `solve-quadratic1` by substituting the solution in the original quadratic form, evaluating it, and returning the result. Examples:

```
(test-quadratic-solver solve-quadratic1 1 2 2)
→ 0
(test-quadratic-solver solve-quadratic1 1 2 3)
→ -4.440892098500626e-16+0.0i
```

Ideally the result of the test, called the *residue*, is 0, but because the built-in function `sqrt` does not always give exact results due to finite precision, the residue may be non-zero.

4. Implement and test an alternative quadratic solver `solve-quadratic2` based on the alternate formula: $-2c/(\text{sqrt}(b^2 - 4ac) + b)$. Compare the results for the two solvers for a variety of coefficients. [To read further about why the variants differ, consult Section 5 of "*Pitfalls in Computation, or Why a Math Book Isn't Enough*", by George Forsyth: http://www.jstor.org/view/00029890/di991556/99p1071d/]

```
(solve-quadratic2 1 1e10 1)
→ -1e-10
```

5. Implement and test a third quadratic solver `solve-quadratic` that combines the above two, giving the solution corresponding to the residue of least magnitude. Also this version should handle the case where coefficient *a* can be 0. (Use the built-in function `zero?` to test.) Example:

```
(solve-quadratic 1 1e10 1)
→ -1e-10
(solve-quadratic 0 1 2)
→ -2
```

6. Construct the function `how-many`, which expects two arguments, an item and a list, and counts the number of times the item occurs in the list. Use the built-in function `equal?` for testing equality. It applies to arbitrary elements and not just numbers. Example:

```
(how-many 5 '(3 4 5 7 5 9 3 2 5))
→ 3
```

7. In class, we created a function that returns the prime factors of any positive integer as a list. Example:

```
(factors 24)
→ (2 2 2 3)
```

In this problem, we want to do something similar, but we want the result to be more compact. Specifically, the elements of the resulting list will be 2-lists, the first element being a factor and the second being a multiplicity or exponent. The function is to be called compact-factors. The factor 1 should never appear, because it conveys no information. Examples:

```
(compact-factors 24)
→ ((2 3) (3 1))
(compact-factors 466219353221696151)
→ ((3 11) (7 11) (11 3))
(compact-factors 1)
→ ()
```

It is preferred that you implement from scratch, rather than using the function `factors`.

8. Create the function `un-factor` that computes the number corresponding number from a list of compact factors:

```
(un-factor '((2 3) (3 1)))
→ 24
(un-factor '()
→ 1
```

9. Construct the function `number-each` which expects a single argument that is an arbitrary list, say L. It creates a list of 2-lists, where each 2-list consists of an ordinal number, starting with 1, followed by the corresponding element of L. Examples:

```
(number-each '(jan feb mar apr)))
→ ((1 jan) (2 feb) (3 mar) (4 apr))
(number-each '(I II III IV V VI)))
→ ((1 I) (2 II) (3 III) (4 IV) (5 V) (6 VI))
(number-each '())
→ ()
```

Hint: Use a more general auxiliary function.

10. In class, you learned about *association lists* (A-lists for short). A common use of such a list is to represent a function, namely the function that finds the second element of the first 2-list that has a first element equal to the function's argument. The built-in function `assoc` will find the first 2-list in a list, the second argument, such that its first element is equal to the first argument. For example:

```
(assoc 3 '((1 jan) (2 feb) (3 mar) (4 apr)))
→ (3 mar)
```

The convention is that if there is no such element in the 2-list, `assoc` returns `#f`. Since `#f` cannot be confused with a 2-list, the meaning of this result is clear. In this problem we want to use an A-list to score Scrabble-words. The following Scheme statement defines an A-list that provides all of the letter scores:

```
(define scrabble-letter-scores
'((#\a 1) (#\b 3) (#\c 3) (#\d 2) (#\e 1) (#\f 4) (#\g 2)
(#\h 4) (#\i 1) (#\j 8) (#\k 5) (#\l 1) (#\m 3) (#\n 1)
(#\o 1) (#\p 3) (#\q 10)(#\r 1) (#\s 1) (#\t 1)
(#\u 1) (#\v 4) (#\w 4) (#\x 8) (#\y 4) (#\z 10)))
```

The notation '`#\a` is Scheme's way of introducing character literals (and `|#a|` is the way in which characters print).
Using `assoc`, construct a function `score-letter` that determines the score for its letter as an argument. Example:

```
(score-letter '#\w)
→ 4
```

Define a function `score-word` that determines the score for a string of letters. Note that strings in Scheme are not the same as symbols. Strings are doubly quoted. Example:

```
(score-word "foo")
→ 6
```

If possible, we'd like you to use the higher-order functions `map` and `foldr` to do this, rather than recursion.

Don't turn in this week, but thinking ahead to next week: Define a function `best-word` that takes a string representing a set of letters and a list of strings representing allowable words, and which returns a word and its score in the list that can be constructed with letters in the set and which has a maximal score among all such words. (If there is a tie, it returns one of the words arbitrarily.) In grading your problems, we will make sure that the word in each test case is unique.

```
(best-word "academy"
(list "ace" "ade" "cad" "cay" "day"))
→ ("cay" 8)
```

Use this form of documentation for your file, solutions, and functions.
This is mandatory, in the sense that points will be deducted for infractions.
Use plenty of **whitespace** so that your solutions are readable.
Be generous with comments. You need not go quite to the extreme below, making comments on very obvious formulaic computation, but make sure your method is clear.
Dr. Scheme will automatically indent for you, but it might be a good idea to run the command **Reindent All** prior to submission. Also, Dr. Scheme will not reassign lines.

```
;; Christine Alvarado
;; CS 60 assignment 1
;; Problem 0: Factorial function
;;
;; Inputs:
;; x, an integer
;;
;; Output:
;; the factorial of x
;;
;; Method:
;; The input, x, is multiplied by
;; the factorial of (x-1), unless it happens
;; to be less than 2, in which case 1 is returned.

(define (fact x)
  (if (< x 2)
      1
     (* x (fact (- x 1)))))
```